NASA Technical Memorandum 103951

Parallel Implementation of an Algorithm for Delaunay Triangulation

Marshal L. Merriam
Ames Research Center, Moffett Field, California

July 1992

NASA
National Aeronautics and Space Administration

Ames Research Center
Moffett Field, California 94035-1000

SUMMARY

This work concerns the theory and practice of implementing Tanemura's algorithm for 3D Delaunay triangulation on Intel's Gamma prototype, a 128 processor MIMD computer. Efficient implementation of Tanemura's algorithm on a conventional, vector processing, supercomputer is problematic. It does not vectorize to any significant degree and requires indirect addressing. Efficient implementation on a parallel architecture is possible, however. In this work, speeds in excess of 20 times a single processor Cray Y-MP are realized on 128 processors of the Intel Gamma prototype.

1. INTRODUCTION

Delaunay triangulation is used for a number of 2D and 3D applications in CFD including grid generation and surface reconstruction (refs. 1 and 2). Triangulation speed has been an issue however, especially for very large meshes. The algorithm discussed here is a 3D version of Tanemura's algorithm, an advancing front method for Delaunay triangulation (refs. 3 and 4). This algorithm can be efficiently implemented on a scalar processor such as a Silicon Graphics IRIS 310/VGX. Efficient implementation on a vector processor such as the Cray Y-MP is difficult, however.

The most time consuming part of the algorithm is a series of range queries (*i.e.*, given (x, y, z) find the nearest of N vertices). An exhaustive search vectorizes easily, first computing the distance to each vertex, then finding the minimum of these through an optimized system routine requiring $O(\ln N)$vector startups and $O(N)$arithmetic. Using a tree search based on quadtrees or (as here) coordinate bisection (ref. 5), reduces the arithmetic complexity to $O(\ln N)$ in the number of vertices, but eliminates vectorization except for a small exhaustive search at the bottom of the tree. This improvement is critical because Tanemura's algorithm requires several range queries per tetrahedron. Usually there are about $7N$ tetrahedra, though in pathological cases (which never happen accidentally) there can be $O(N^2)$of them. Thus, for the average case, the complexity is $O(N^2)$with vectorization or $O(N \ln N)$without it.

The usual vectorization strategy, in cases like this, is to do a number of tree searches at once, in effect vectorizing over the outer loop. In principal, this can be done for Tanemura's algorithm. The strategy involves dividing all the queries according to which branch of the tree search they take. This can be vectorized. Each of the two groups can be further subdivided. This continues until the groups become too small for effective vectorization. The degree of vectorization possible depends on the number of simultaneous range queries and their spatial distribution. There is a degradation, perhaps as much as a factor of two, from the completion of unnecessary range queries. These redundancies do not occur for the sequential algorithm. Because of the significant programming effort required, this strategy was not implemented. Instead, a partitioning strategy was used to effectively take advantage of the MIMD parallelism available through Intel's Gamma prototype.

Under the MIMD paradigm, each processor has a separate copy of the program, and a spatially contiguous portion of the data. Under this domain decomposition approach, each processor is responsible for many range queries. These need not be (and generally aren't) synchronized. The fact that range queries in different processors require different amounts of time is not a problem, since they don't interact. Processors finishing early simply proceed to the next query. Similarly, the differences in the number of queries required to form a given tetrahedron also don't affect efficiency. The remaining problem is controlling the interactions between tetrahedra, especially those on different processors. This is done by appropriate partitioning of the domain.

2. OVERVIEW OF TANEMURAS ALGORITHM

In the interests of clarity, a brief review of three dimensional triangulation, Delaunay triangulation and Tanemura's algorithm is in order. The problem begins with a set of nodes in three dimensions. These are given by their (x, y, z) coordinates. Any proper triangulation tessellates space with tetrahedra in such a way that

1. The vertices of the tetrahedra are the nodes of the mesh,
2. All nodes of the mesh are vertices of one or more tetrahedra,
3. All the nodes are contained within a single polyhedron with triangular facets, and
4. All space within this polyhedron is filled with tetrahedra.

There is a considerable literature on Delaunay triangulation, its definition, its construction, and its properties (see ref. 6 for a survey). In this case the *insphere* property is relevant. It follows from 1–4 above that no tetrahedron contains a node. Delaunay triangulation has a stronger property. The circumsphere of any tetrahedron contains no nodes. An algorithm first shown by Tanemura (ref. 3) makes use of this property to generate the Delaunay triangulation of a set of points.

Each tetrahedron is composed of four triangular faces. It is convenient to represent the connectivity of the finished triangulation as a list of these faces. Each item of a face list is composed of the indices of the three nodes that are its vertices, and the index of the tetrahedra on either side. A list of the faces for each tetrahedra can be constructed from a face list in one pass.

2.1 Making the First Tetrahedron

The algorithm begins by finding a node near the center of the cloud of points. The line connecting this node with its closest neighbor is certain to be an edge of the Delaunay triangulation (ref. 8).

This edge is promoted to a triangle by connecting both endpoints to another node, the node which results in the minimum radius circumcircle for the triangle. In practice,

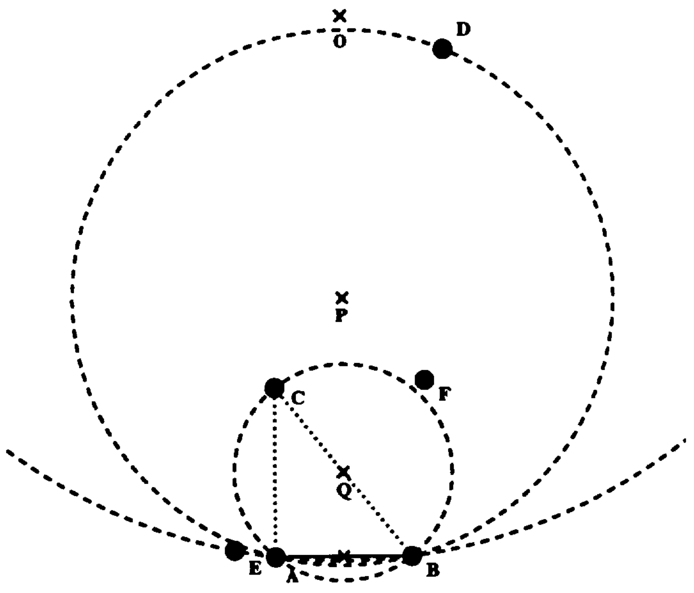

FIGURE 1. In 2D, Tanemura's algorithm promotes edge AB to triangle ABC. Node E is considered first, being the closest to the midpoint of AB. The center of ABE's circumcircle is at point O. Node D is the closest to O. The circumcenter of ABD is at P. Node C is the closest to P. The circumcircle for ABC contains no other nodes, so triangle ABC is formed. The situation in 3D is similar, with minimum circumspheres substituting for circumcircles.

this is done iteratively, starting with the node closest to the midpoint of the edge. If the circumsphere of this proposed triangle (the sphere with center and radius identical to the triangles circumcircle) contains another node, the Delaunay edge is connected to that node instead. This process is repeated until a triangle is found which contains no nodes within its circumsphere. Such a triangle is guaranteed to be a face of the Delaunay triangulation. Figure 1 depicts this process in two dimensions.

Finally the triangle is promoted to a tetrahedron by connecting all three vertices to another node. As before, this is done iteratively, starting with the node closest to the triangle's centroid. If the circumsphere of the proposed tetrahedron contains another node, the Delaunay face is connected to that node instead. This process is continued until a tetrahedron is found which contains no nodes within its circumsphere. From the insphere property we know that such a tetrahedron is part of a Delaunay triangulation.

2.2 Advancing the Front

The initial tetrahedron consists of four faces which make up the initial face list. These faces are only partially specified; they are each missing the index of a neighboring tetrahedron. Such faces are called front faces; collectively they represent a two manifold "front" in three space. Tanemura's algorithm is a frontal method which works by building the missing neighbors. Promoting a front face to a tetrahedron follows the procedure used in building the initial tetrahedron, but with one additional constraint; the node finally selected must lie on the open side of the face, that is, the side opposite the existing tetrahedron.

No such node may exist. In that case the front face is identified as part of the enclosing polyhedron (the convex hull for Delaunay triangulations). The missing neighbor is symbolically identified (perhaps with a negative tetrahedron number) and the face is removed from the front.

The connection of three face vertices to a node creates a new tetrahedron. This completes the information associated with that face and removes it from the front. However, three triangles are defined in the process. If any of these triangles are already front faces, the new tetrahedron is the missing neighbor and another triangle is removed from the front. If not, they become part of the front. The front may expand or contract in this way. The algorithm continues to build tetrahedra in this fashion until the front vanishes.

3. PARALLEL CONSIDERATIONS

In the MIMD paradigm, it is only necessary to eliminate the fine scale data dependencies between processors. Synchronization takes place much less often, so statistical load balancing strategies can be effective. In this case the principal data dependencies come from neighboring front faces. If the two faces form a local concavity (*i.e.*, if their common edge is a reflex edge of the front) then they are likely to be part of the same tetrahedron. It would be redundant to build tetrahedra on both simultaneously. Furthermore, it would be a possible source of synchronization problems and related errors.

To prevent this occurrence and allow massive parallelism, the nodes have been divided into physically disjoint sets each of which has its own front (domain decomposition). The front can be characterized as a number of triangular faceted polyhedra. If each such polyhedron is completely contained within a single processor, there can be no data dependency of this type.

When faces from two different processors meet at an edge, steps must be taken to avoid building on both faces at once. The approach taken here is to ensure that only one face from each such pair can be built on. In this implementation the arbitrary choice is based on a global numbering scheme described in section 3.2 below.

3.1 Partitioning

The input for Delaunay triangulation is a collection of nodes in 3-space, specified by their coordinates, (x, y, z). These are arbitrarily distributed among the various processors in such a way that each processor has an equal (or nearly equal) number of nodes. The nodes are then redistributed using a coordinate bisection strategy, implemented in parallel as described below.

The mean x coordinate of all the the nodes is determined, and the nodes within each processor are divided into two approximately equal groups, according to whether the x coordinate is smaller than the mean or not. Nodes are exchanged between processor pairs whose binary address differs in the most significant digit. The low numbered processors get the nodes with x coordinates less than the mean.

If there are more than two processors, this partitioning is repeated, with the y coordinate, for each half of the cube. A separate mean y is used for those processors with low numbers and those with high numbers. This divides the nodes more or less equally among the four groups of processors.

Continuing along these lines, partitioning is next done in z, then x again and so on until each partition is composed of a single processor. This partitioning of the nodes results in a nearly equal number of nodes in each processor. It also results in a well defined partition of the domain. A given (x, y, z) location is associated with a unique processor.

3.2 Numbering System

Both a local and a global numbering scheme are used to identify vertices, faces, and tetrahedra. For ease and efficiency of programming, a local numbering scheme is desired, with each processor having nodes numbers starting from 1. On the other hand, a global numbering is required, if for no other reason than to uniquely identify each node in the resulting triangulation. For this reason, each node has a global number N_G, a local number N_L and processor number N_P. A node can be uniquely identified either by its global number, or by the combination of its processor number and its local number.

When a triangle's vertices are all on the same processor (in the same partition), the triangle is kept on that processor and the local numbers are used. When a triangle's vertices are not all on the same processor, a local numbering scheme is only valid for nodes that reside on the processor where the triangle is kept. Other vertices must somehow incorporate both their processor number and the local node number on that processor. This is done by simply taking a linear combination of the two numbers to form a composite node number N_C defined as follows:

$$N_C = 1000000 N_P + N_L \tag{1}$$

For this purpose, processor numbers start at 1. Composite numbers are easily decoded to find N_P and N_L.

Residence of triangles (and tetrahedra) that straddle processor boundaries is arbitrated using global node numbers. It is the processor where the vertex with the lowest global node number resides.

One more special numbering convention is used. Sometimes, when promoting a triangle to a tetrahedron, the node selected will reside on another processor. If the resulting tetrahedron would reside on the other processor, it is not formed. Instead, the neighbor number

$$N_N = -1000000 N_P \tag{2}$$

is used as the tetrahedron number. Later, the tetrahedron will be formed from a face on the other processor (the residence of a tetrahedron is the residence of at least three of its four faces), and the true tetrahedron number will be filled in.

4. COMMUNICATION

Within a particular processor, the partitioning continues until only a few nodes (typically 18) occupy each partition. This tree structure allows efficient range queries within a single processor. Typically, these queries ask for the node closest to a particular (x, y, z) location. In most cases a definitive response can be formulated using only the information from a single processor. In some cases though, neighboring processors will have to be consulted. It is here that the difference between a shared memory architecture (like the Cray-2) and a message passing architecture becomes most apparent.

In a shared memory architecture, data contained on other processors is directly addressable. For this particular query, there is no data corruption problem because the coordinates of the vertices never change.

In a message passing architecture, one processor cannot directly fetch data from the memory of another. Furthermore, in a tree structured search like this, it may not be obvious which data needs to be fetched. The approach used here involves a query and response. The relevant (x, y, z) coordinates are passed, and information about the closest node to it is returned.

Although the Intel allows interrupt driven code, this was not used. Instead, queries from foreign processors were honored at the end of each iteration, a procedure somewhat akin to reading your mail at the end of the day. As written, an off processor query halts execution until the response is received. If naively implemented, this could result in deadlock, where two processors wait on each other, neither able to finish the iteration. The solution chosen here is to process foreign queries while waiting for a response.

Several specific types of messages were used. Five are listed here. Note that the first is an informational message in which no direct response is expected. The others are query/response pairs.

iamdone– A message indicating that a particular processor is out of work, except for fielding queries from other processors. When all processors are out of work, its time to proceed to the output phase. This has the same effect as a global synchronization except that queries from other processors can be processed while waiting.

nodesend– A message requesting a range query from among the vertices contained on a particular processor. The message contains the locations of the three front vertices in question, the best current guess for the fourth vertex, and some information about constraints, if any. This is enough to decide if any of the nodes on a particular processor are an improvement over the current best guess and, if so, which one.

noderecv– A message stating which node is the best. This is the reply to nodesend. Also given are the coordinates of the best node and the size of the circumsphere containing it and the front triangle.

facesend– It is possible for a face to be created by a process that doesn't own it. In this case, the face is sent to the other processor to be added to the face list on that processor.

facerecv– This is returned for synchronization only. First in, first out is implicitly assumed in several places and a handshake is one way to be sure this is true.

4.1 Avoiding Communication

The majority of communication takes the form of range queries to other processors. Each processor contains points from a particular region in space and the bounding boxes for these regions do not overlap. This information can sometimes be used to avoid communication.

For example, the local processor is first searched to find the node which is closest to the desired point. Usually such a node exists and is a certain distance from the desired point. Using bounding box information, it can sometimes be established that any nodes contained in another processor are farther away. The need to search a foreign processor can thus be avoided by establishing that the search will be unsuccessful. Significant efficiency gains result.

5. IMPORTANT DETAILS

In spite of the apparent simplicity of the implementation described above, completeness requires correct treatment of two possible problems. These occur rarely; some point sets don't need them at all. They are implemented as a sort of postprocessing step after all the other triangulation is finished.

The first of these problems comes about when a processor finishes early. Through the routine handling of queries and responses, a new face may be added to its face list. This means the processor is no longer done. Other processors, needed to process any queries, think it is done, and may quit at any time, so resuming triangulation is unsafe. The procedure here is to wait for synchronization and then process the few "left over" faces of this kind.

Another problem is the persistence of small regions that are not tetrahedralized. Even though ownership of a tetrahedron guarantees ownership of three out of its four faces, it

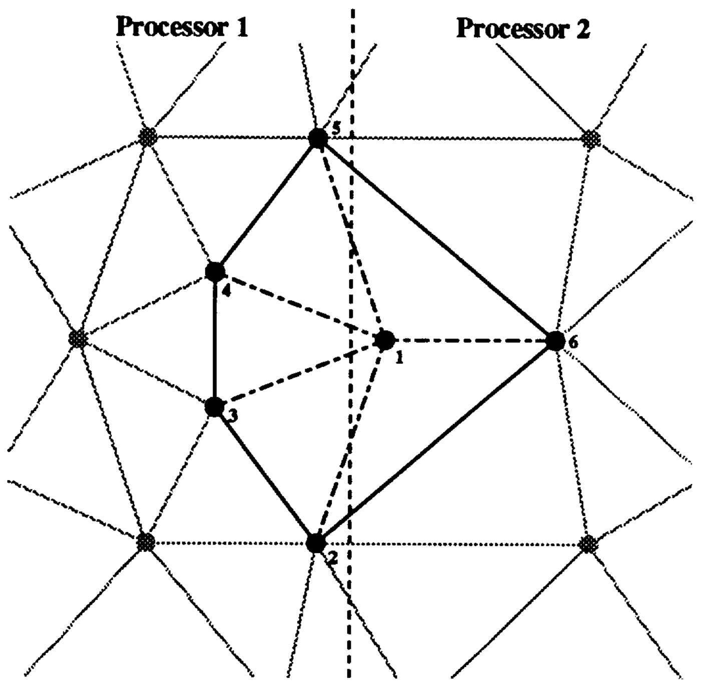

FIGURE 2. The global node numbers are such that the region shown is never triangulated. All five triangles reside in processor 2, but they are surrounded by edges residing in processor 1. An analogous situation occurs in 3D.

is possible for a group of tetrahedra owned by one processor to be completely surrounded by triangles belonging to other processors. A 2D example of this situation is shown in figure 2.

This situation can be detected by the existence of neighbor numbers N_N in places where tetrahedron numbers would be expected. It is handled by sending a description of the missing tetrahedra to the processors where they reside.

6. RESULTS AND CONCLUSIONS

The serial code is operational and has been used on cases of up to 500,000 nodes. The platform of choice for the serial code is the IRIS 310/VGX, using large amounts of virtual memory. Speeds of 40 - 120 nodes per second were measured, the bigger problems slowing down because of page faults and a log term in the complexity. Porting the code to Cray Y-MP (1 processor) reveals two things. The speed is only about 2-3 times the speed of the workstation, and the memory available to the user is considerably less. The Cray is a shared machine which significantly degrades turnaround time and memory availability. Use of performance monitoring utilities on the Cray confirm that very little of the code is vectorized, in spite of considerable efforts to do so. Indirect addressing, conditional execution, and considerable integer arithmetic further degrade performance. The measured floating point performance was about 7 Mflops, a figure that indicates how little of the

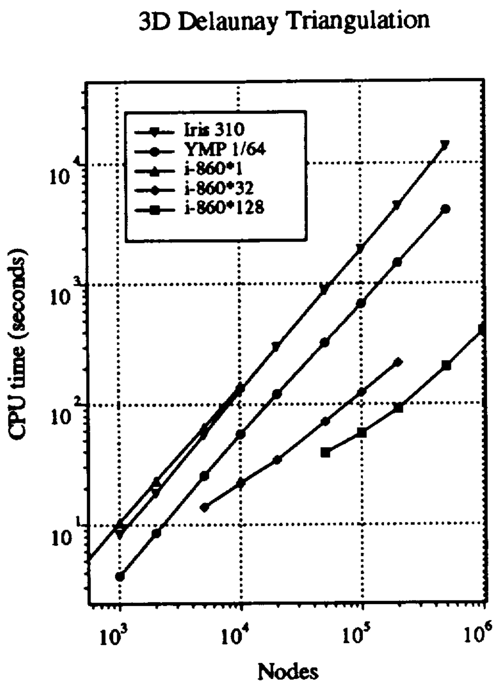

FIGURE 3. Performance of Tanemura's algorithm. The 128 processor performance of Intel's Gamma prototype is 20 times as fast as the Y-MP for this problem. The degradation for smaller problems is due to relatively higher communication costs.

code involves floating point calculations, and how little of that is vectorized. Higher Mflop numbers were obtained, but only by doing redundant work – execution times were not improved.

The parallel code has been implemented on the Intel Gamma Prototype. The results are summarized in figure 2 for the case where the triangulation sites are randomly distributed inside a unit cube. The single processor speed of the i860 is about that of the IRIS for this problem, though memory limitations keep the maximum size case down to about 13000 nodes. Virtual memory is not available at this time.

Memory is sufficient to accommodate a 1,750,000 node triangulation in 128 processors, though the largest case actually run was 1 million nodes, executing in about 7 minutes, more than 20 times Cray speed. It was also twice the size of the largest Cray job, mostly due to the difference in integer precision (32 bits on the i860 versus 64 bits on the Y-MP).

Doubling the number of processors does not double the speed, due to various overheads including communication and I/O. The overhead is not constant but appears to increase slightly with the number of nodes. This essentially sequential behavior is explained by the disk bandwidth. Though large, the bandwidth does not depend on the number of processors, so reading in the problem takes a certain irreducible amount of time. On the largest cases the I/O time accounts for 2% of the run time. The effect is proportionately higher for some small cases as the disk latency becomes significant. If some care is not paid to the I/O strategy, it can become the dominant cost. Direct access, unformatted reads (one per processor) were used here to realize as much as possible of the available disk bandwidth. By contrast, sequential access, formatted reads takes about 30 times as long, since all I/O goes through a single processor and must be converted from ASCII to binary. Direct access, formatted reads take about 300 times as long. Short records and no look-ahead mean that the disk latency determines the read time.

A second possible source of overhead is the formation of the convex hull. To establish that a face is part of the convex hull requires that a particular half space be clear of all nodes. As originally implemented, this required checking every node (on every processor). A more enlightened implementation simply checks the corners of the bounding box for each processor. This has dramatically reduced the amount of communication, as well as the overall algorithmic complexity.

Communication between processors is important but depends mainly on the number of nodes per processor, a bounded quantity. This term accounts for the improvement in speed relative to the Cray, as problem size increases. For example, in figure 3, the 32 processor results show only a 45% improvement in run time over those for the Cray when 5000 nodes are triangulated, but this grows to more than a factor of 6 with 200K nodes. A similar effect is seen on the 128 processor results.

A minor effect which is not accounted for in figure 3 is the time required to load the program. This varied between 15 and 30 seconds, depending on the presence of other users.

There is considerable room for optimism here. It seems likely that the i860 will improve in speed as the compiler technology catches up with that used elsewhere. It should be 20% faster than an IRIS workstation just based on the clock rate. Recent experience with Vectoral compilers for the i860 chip (ref. 7) indicate that improvements of a factor of 3 are possible in this area, mostly through instruction scheduling improvements. The communication strategy used here is early in its development cycle and significant improvements are expected in the future. In particular, strategies which combine messages into fewer, larger ones, promise to reduce communication time through better amortization of the latency.

On balance, this implementation of Tanemura's algorithm has produced a practical and efficient way of triangulating very large numbers of points. It is of special interest because it utilizes a true MIMD paradigm. As such, it is fundamentally different than many other parallel implementations, which can be (and often are) efficiently implemented on SIMD machines.

7. REFERENCES

1. Baker, T. J.: Three Dimensional Mesh Generation by Triangulation of Arbitrary Point Sets. AIAA Paper 87-1123-CP, 1987.

2. Maksymiuk, C.; and Merriam, M. L.: Surface Reconstruction from Scattered Data Through Pruning of Unstructured Grids. AIAA Paper 91-1584, 10th CFD Conference, June 24-27, 1991.

3. Tanemura, M.; Ogawa, T.; and Ogita, N.: A New Algorithm For Three Dimensional Voronoi Tessellation. Journal of Computational Physics, vol. 51, 1983, pp. 191–207.

4. Merriam, M.: An Efficient Advancing Front Algorithm For Delaunay Triangulation. AIAA Paper 91-0792, 29 Aerospace Sciences Meeting, Reno, Jan. 7-11, 1991.

5. Bentley, J. L.: Multidimensional Binary Search Trees Used For Associative Searching. Communications of the ACM, vol. 18, no. 9, Sept. 1975, pp. 509–517.

6. Bern, M. W.; and Eppstein, D.: Mesh Generation and Optimal Triangulation. Xerox PARC Technical Report, CSL-92-1.

7. Wray, A.; and Rogallo, R.: Simulation of Turbulence on the Intel Gamma and Delta. NASA TM, 1992 *To appear.*

8. Preparata, F. P.; and Shamos, M. I.: Computational Geometry, an Introduction. Springer Verlag, New York, 1985.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE July 1992	3. REPORT TYPE AND DATES COVERED Technical Memorandum

4. TITLE AND SUBTITLE Parallel Implementation of an Algorithm for Delaunay Triangulation	5. FUNDING NUMBERS 505-59-53
6. AUTHOR(S) Marshal L. Merriam	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Ames Research Center Moffett Field, CA 94035-1000	8. PERFORMING ORGANIZATION REPORT NUMBER A-92144
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001	10. SPONSORING/MONITORING AGENCY REPORT NUMBER NASA TM-103951

11. SUPPLEMENTARY NOTES
Point of Contact: Marshal L. Merriam, Ames Research Center, MS 202A-1, Moffett Field, CA 94035-1000
(415) 604-4737

12a. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified — Unlimited Subject Category 59	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This work concerns the theory and practice of implementing Tanemura's algorithm for 3D Delaunay triangulation on Intel's Gamma prototype, a 128 processor MIMD computer. Efficient implementation of Tanemura's algorithm on a conventional, vector processing, supercomputer is problematic. It does not vectorize to any significant degree and requires indirect addressing. Efficient implementation on a parallel architecture is possible, however. In this work, speeds in excess of 20 times a single processor Cray Y-MP are realized on 128 processors of the Intel Gamma prototype.

14. SUBJECT TERMS Parallel computing, Triangulation, Grid generation	15. NUMBER OF PAGES 13
	16. PRICE CODE A02

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT